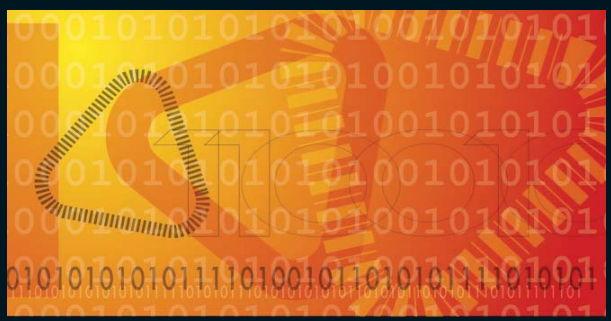
CodeFlow With Multithreaded Support



Want your multithreaded programs supported by a CodeFlow tool? Read on! This article will extend the CodeFlow tool to trace the programs that are spawned by multiple threads.

Last month we worked out how to develop a simple, lightweight, yet powerful tool called CodeFlow. This is used to trace programs using the instrumentation option provided by the GNU gcc compiler. But one of the important features that is still missing is support for CodeFlow from multithreaded programs. In this article we will explore how to extend the CodeFlow tool so that it can trace even the programs that spawn multiple threads.

A thread (also called an *execution context* or *lightweight process*) is a single sequential flow of control within a program. Multithreaded programs usually spawn one or more threads so that more than one activity can be performed in parallel, within the same application. When the CodeFlow tool (without multithread support) is used to trace such programs, the report created will be of no use, as it will be interleaved with trace information from different threads spawned by the traced program. To overcome this, we need to modify our CodeFlow tool so that it can create and maintain trace information for each thread, separately.

Figure 1 shows a simple multithreaded program, which we will trace using our modified CodeFlow tool.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <pthread.h>
5
6  void thread_handler_display() {
7      printf("%s:%d:\n",__FUNCTION__,pthread_self());
8  }
9
10 void *thread_handler1(void *data) {
11     pthread_t tid = pthread_self();
12     printf("%s: Thread (%d) started.\n",__FUNCTION__,tid);
13     thread_handler_display();
14     printf("%s: Thread (%d) ended.\n",__FUNCTION__,tid);
15     return(NULL);
16 }
17
18 void *thread_handler2(void *data) {
19     pthread_t tid = pthread_self();
20     printf("%s: Thread (%d) started.\n",__FUNCTION__,tid);
21     thread_handler_display();
22     printf("%s: Thread (%d) ended.\n",__FUNCTION__,tid);
23     return(NULL);
24 }
25
26 void display1() {
27     printf("%s:%d:\n",__FUNCTION__,pthread_self());
28 }
29
30 void display() {
31     printf("%s:%d:\n",__FUNCTION__,pthread_self());
32     display1();
33 }
34
35 int main(int argc, char*argv[]) {
36   pthread_t tid, tid1, mtid;
37   void *ret = NULL;
38
39   mtid = pthread_self();
40   printf("%s:Main thread %d started.\n",__FUNCTION__,mtid);
41
42   display();
43
44   pthread_create(&tid,NULL,thread_handler1,NULL);
45   pthread_create(&tid1,NULL,thread_handler2,NULL);
46
47   pthread_join(tid,&ret);
48   pthread_join(tid1,&ret);
49
50   printf("%s:Main thread %d ended.\n",__FUNCTION__,mtid);
51   return(0);
52 }
```

Figure 1: A simple multithreaded program

In Figure 1 you can see that the program spawns two threads: *thread_handler1* and *thread_handler2* (line number 44 and 45), in addition to the main thread. The pthread API *pthread_self* is used to get the identifier for the current thread.

Figure 2 shows the modified header file for the CodeFlow tool.

```
1  /*
2  *
3  * codeflow.h - Header file for CodeFlow for C/C++ programs
4  *
5  * Raja R.K
6  *
7  */
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <pthread.h>
12
13 #define __NIF__ __attribute__((__no_instrument_function__))
14 #define __FUNC_ENTER__ __cyg_profile_func_enter
15 #define __FUNC_EXIT__ __cyg_profile_func_exit
16 #define TRACE_FILE "trace.out"
17 #define MAX_THREADS 500
18
19 typedef struct threadInfo {
20     pthread_t tid;
21     long indent;
22     FILE *fptr;
23 } threadInfo;
24
25 threadInfo tInfo[MAX_THREADS];
26 int tIndex = 0;
27 int first_time = 1;
28
```

Figure 2: Header file for CodeFlow with multithreaded support

In Figure 2, the macro *MAX_THREADS* defines the maximum number of threads that CodeFlow can trace. The *threadInfo* structure holds the thread identifier, trace file and indent information for each thread spawned by the traced program.

Figures 3 and 4 show the implementation of the modified CodeFlow tool.

In Figure 3 you can see that the pthread API *pthread_self* is used to get the identifier for the current thread (line 25). The

thread identifier and indent information for each thread are now maintained separately (line 28 and 29).

```
1  /*
2  *
3  * codeflow.c - CodeFlow for C/C++ programs with multithreaded support
4  *
5  * Raja R.K (rajark_hcl@yahoo.co.in)
6  *
7  */
8
9  #include "codeflow.h"
10
11 void __NIF__ exit_handler()
12 {
13     int i = 0;
14
15     for( i=0; i<MAX_THREADS; i++ ) {
16         if( tInfo[i].fptr != NULL ) {
17             fclose(tInfo[i].fptr);
18         }
19     }
20 }
21
22 void __NIF__ __FUNC_EXIT__(void *this_fn, void *call_site)
23 {
24     int i = 0;
25     pthread_t tid = pthread_self();
26
27     for( i=0; i<MAX_THREADS; i++ ) {
28         if( tInfo[i].tid == tid ) {
29             tInfo[i].indent--;
30             break;
31         }
32     }
33 }
34
```

Figure 3: CodeFlow with multithreaded support

```
35 void __NIF__ __FUNC_ENTER__(void *this_fn, void *call_site)
36 {
37     int i = 0, j = 0, k = 0, found = 0;
38     pthread_t tid = pthread_self();
39     char fname[512];
40
41     if( first_time ) {
42         atexit(exit_handler);
43         first_time = 0;
44     }
45
46     for( i=0; i<MAX_THREADS; i++ ) {
47         if( tInfo[i].tid == tid ) {
48             found = 1;
49             break;
50         }
51     }
52
53     if( !found ) {
54         sprintf(fname,"codeflow.%d.trace",tid);
55         if( (tInfo[tIndex].fptr=fopen(fname,"w")) == NULL ) {
56             return;
57         }
58         printf("Program trace for thread %d will be saved in %s\n",tid,fname);
59
60         tInfo[tIndex].tid = tid;
61         tInfo[tIndex].indent = 0;
62         j = tIndex;
63         tIndex++;
64     } else {
65         j = i;
66     }
67
68     for( k=0; k<tInfo[j].indent; k++ ) {
69         fprintf(tInfo[j].fptr,"\t");
70     }
71
72     fprintf(tInfo[j].fptr,"%x\n",this_fn);
73     tInfo[j].indent++;
74     fflush(tInfo[j].fptr);
75 }
```

Figure 4: CodeFlow with multithreaded support

In Figure 4 you can see that at line 42, the exit handler is registered to close all the open file streams at the end of the trace. The function then retrieves the thread information for the current thread (line 46 to 51). If the thread information could not be found, then a new entry is created (line 53 to 64), and the thread info structure is updated with the thread identifier, trace file and indent information (line 68 to 73) of the current thread. You can see that in our modified version of the CodeFlow tool, a new trace file is created for every thread spawned by the traced program.

```
1  #
2  # codeflow_report.sh - Report generation script for CodeFlow
3  #
4  # Raja R.K (rajark_hcl@yahoo.co.in)
5  #
6  #
7
8  #!/bin/sh
9
10 TRACE_FILE="$2"
11 TRACE_TMP1_FILE="$2.tmp.1"
12 TRACE_TMP2_FILE="$2.tmp.2"
13 PRG="$1"
14 NM="/usr/bin/nm"
15 TR="/usr/bin/tr"
16 CUT="/bin/cut"
17 CP="/bin/cp"
18 GREP="/bin/grep"
19 SED="/bin/sed"
20 MV="/bin/mv"
21 RM="/bin/rm"
22 SYM_FILE="$PRG.sym"
23
24 $NM -g -l --defined-only $PRG | $TR -s " " "^" | $TR -s "\t" "^" | $CUT -f1,3,4 -d"^" > $SYM_FILE
25
26 $CP $TRACE_FILE $TRACE_TMP1_FILE
27
28 cat $TRACE_FILE |
29 while read line
30 do
31     SYM_NAME=`$GREP "$line" "$SYM_FILE" | $CUT -f2 -d"^"`
32     $SED -e s/"$line"/"$SYM_NAME"/ $TRACE_TMP1_FILE > $TRACE_TMP2_FILE
33     $MV $TRACE_TMP2_FILE $TRACE_TMP1_FILE
34 done
35
36 $MV $TRACE_TMP1_FILE $TRACE_FILE
37 $RM -rf $SYM_FILE
38 echo -e "CodeFlow report saved in: $TRACE_FILE"
39
```

Figure 5: CodeFlow report script

Figure 5 shows the modified version of the CodeFlow report script. Line 10, 11 and 12 are modified to accept the trace file as an argument from the user.

Now let us see how we could use our modified CodeFlow to get the trace (call graph) for the sample program shown in Figure 1.

Step 1: Compile *codeflow.c*

```
gcc -g -O -c codeflow.c
```

Step 2: Compile and link the sample program with CodeFlow...

```
gcc -g -finstrument-functions -O -o test1 test1.c codeflow.o -lpthread
```

Step 3: Execute the sample program

```
# ls
codeflow.c  codeflow.o          test   test1.c
codeflow.h  codeflow_report.sh  test1  test.c
# ./test1
Exit handler registered.
Program trace for thread 8192 saved in codeflow.8192.trace
main:Main thread 8192 started.
display:8192:
display1:8192:
display11:8192:
Program trace for thread 8194 saved in codeflow.8194.trace
thread_handler1: Thread (8194) started.
thread_handler_display:8194:
thread_handler1: Thread (8194) ended.
Program trace for thread 16387 saved in codeflow.16387.trace
thread_handler2: Thread (16387) started.
thread_handler_display:16387:
thread_handler2: Thread (16387) ended.
main:Main thread 8192 ended.
#
```

Figure 6: Tracing with CodeFlow

Step 4: Create a CodeFlow report like this...

```
codeflow_report.sh <prg_name> <trace_file>
```

Example:

```
codeflow_report.sh test1 codeflow.8194.trace
```

Step 5: View CodeFlow report.

```
thread_handler2
        thread_handler_display
```

Figure 7: CodeFlow report of thread 2

```
main
        display
                display1
                        display11
```

Figure 8: CodeFlow report for main thread

```
thread_handler1
        thread_handler_display
```

Figure 9: CodeFlow report for thread 1

Figures 7, 8 and 9 show the CodeFlow report for each thread spawned by the traced program.

Figure 7 shows that the *thread_handler2* function calls the *thread_handler_display* function.

In Figure 8 you can see that the *main* function calls *display*, which in turn calls *display1* and *display11*.

In Figure 9 we see that the *thread_handler1* function calls the *thread_handler_display* function.

With multithread support in place, our CodeFlow can now be used to trace any program, no matter how complex it is!!! LFY

By: Raja R K. The author is a lead engineer working with HCL Technologies (Cisco Systems Offshore Development Centre) in Chennai. He can be contacted at: rajark_hcl@yahoo.co.in